The IBM Quantum Computing Challenge Lab 0 | Lab 1

By: Jack Sullivan, Chris Choi, Robert Wu

What is the IBM Quantum Computing Challenge

“Welcome to the IBM Quantum Challenge, the annual code challenge focused on how to use Qiskit. Whether you’re a newcomer or a seasoned veteran, there is something here for you.

The Quantum Challenge consists of a series of Jupyter notebooks that contain tutorial material, code examples, and auto-graded coding challenges for you to fill in. We call each of these notebooks a ‘lab’. “

Lab 0: Overview

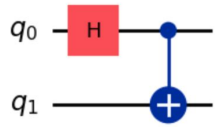
The workflow of Qiskit pattern: Set up, Optimize, Run, and Process your circuit.

Create a two-qubit Bell state and show that it is properly entangled by visualizing the operators.

Step 1: Map our Circuits and Operators

Circuit

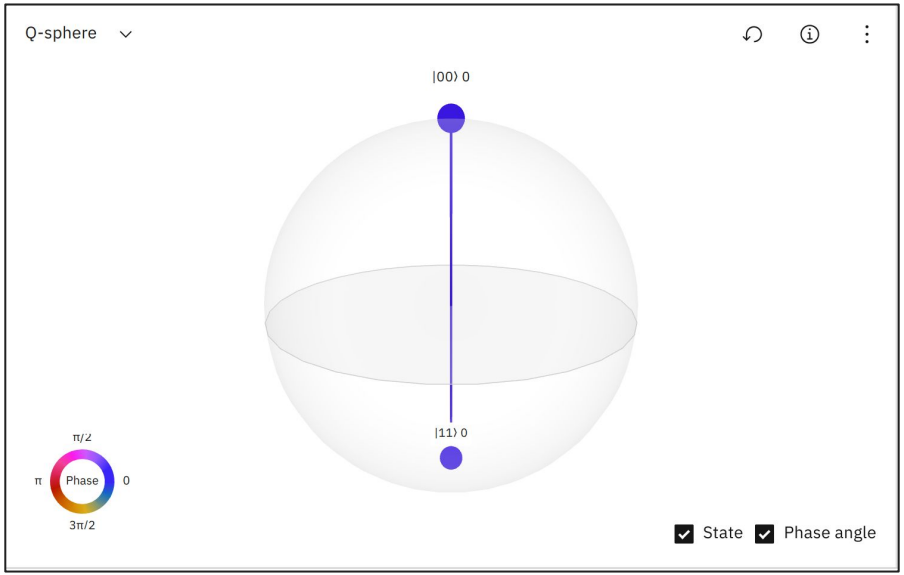
```
# Create a new circuit with two qubits
qc = QuantumCircuit(2)

# Add a Hadamard gate to qubit 0
qc.h(0)

# Perform a CNOT gate on qubit 1, controlled by qubit 0
qc.cx(0, 1)

# Return a drawing of the circuit using MatPlotLib ("mpl"). This is the
# last line of the cell, so the drawing appears in the cell output.
qc.draw("mpl")
```



Operators

```
# The ZZ applies a Z operator on qubit 0, and a Z operator on qubit 1
ZZ = SparsePauliOp('ZZ')

# The ZI applies a Z operator on qubit 0, and an Identity operator on qubit 1
ZI = SparsePauliOp('ZI')

# The IX applies an Identity operator on qubit 0, and an X operator on qubit 1
IX = SparsePauliOp('IX')



### Write your code below here ###

IZ = SparsePauliOp('IZ')

XX = SparsePauliOp('XX')

XI = SparsePauliOp('XI')
### Follow the same naming convention we used above



## Don't change any code past this line, but remember to run the cell.

observables = [IZ, IX, ZI, XI, ZZ, XX]
```

Bell State



Step 2: Optimize

Optimization is done using transpliers.

They are able to reduce the number of gates required, and ensure the structure of gates will work on the machine being used.

Step 3: Execute

Estimator: A function (primitives) that estimates the output of a quantum circuit

Aer Simulator: A fake quantum computer backend

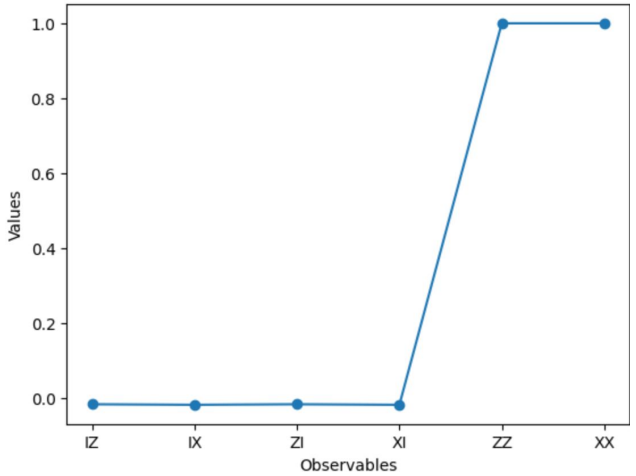
qc: Our quantum circuit

observables: Our operations to run on the circuit

```
[24] # Set up the Estimator
     estimator = Estimator(backend=AerSimulator())

     # Submit the circuit to Estimator
     pub = (qc, observables)

     job = estimator.run(pubs=[pub])
```

Step 4: Post Process the results

```
# Collect the data
data = ['IZ', 'IX', 'ZI', 'XI', 'ZZ', 'XX']
values = job.result()[0].data.evs

# Set up our graph
container = plt.plot(data, values, '-o')

# Label each axis
plt.xlabel('Observables')
plt.ylabel('Values')

# Draw the final graph
plt.show()
```



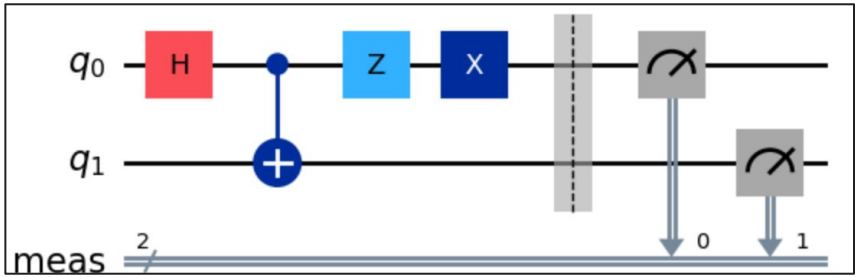
Lab 1: Overview

Qiskit states, the new and the old

VQE with Qiskit 1.0

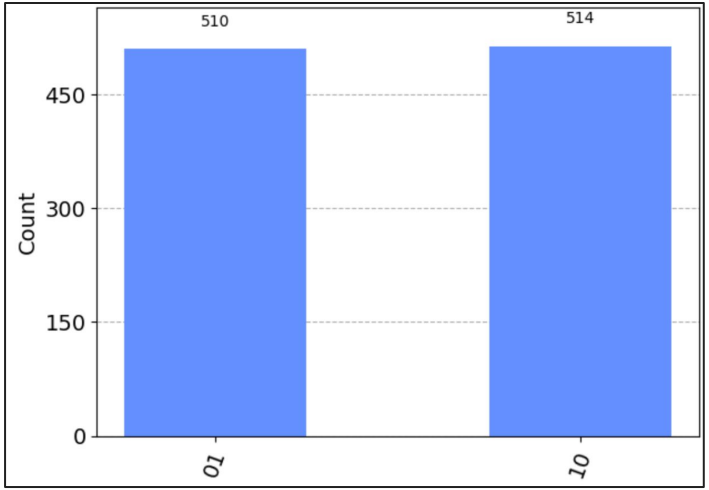
Step 1: Create and Draw a singlet Bell state Circuit

```
# Build a circuit to form a psi-minus Bell state
# Apply gates to the provided QuantumCircuit, qc

qc = QuantumCircuit(2)

### Write your code below here ###

qc.h(0)

qc.cx(0,1)

qc.z(0)

qc.x(0)



### Don't change any code past this line ###
qc.measure_all()
qc.draw('mpl')
```



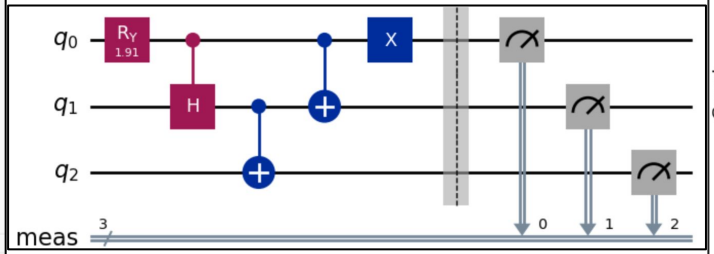
Step 2: Run it with Sampler

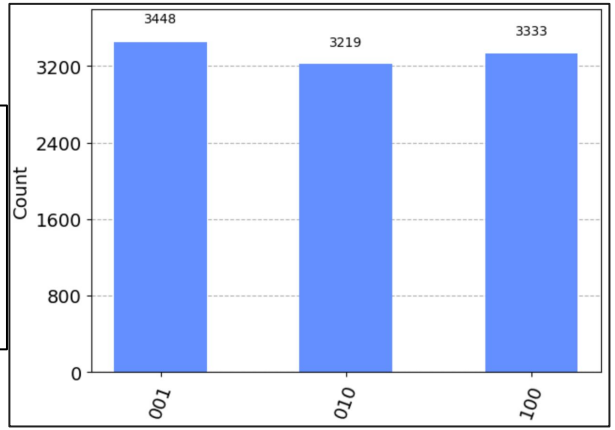
```
qc.measure_all()

### Write your code below here ###


sampler = StatevectorSampler()
pub = (qc)
job_sampler = sampler.run([pub])


### Don't change any code past this line ###

result_sampler = job_sampler.result()
counts_sampler = result_sampler[0].data.meas.get_counts()

print(counts_sampler)
```



Step 3: Create and draw a W-state circuit

```
# Step 1
qc = QuantumCircuit(3)

# Step 2 (provided)
qc.ry(1.91063324, 0)

# Add steps 3-6 below

qc.ch(0,1)

qc.cx(1,2)

qc.cx(0,1)

qc.x(0)

### Don't change any code past this line ###
qc.measure_all()
qc.draw('mpl')
```





Dive into VQE

VQE: Variational Quantum Eigensolver.

Executing a VQE algorithm requires these three steps:

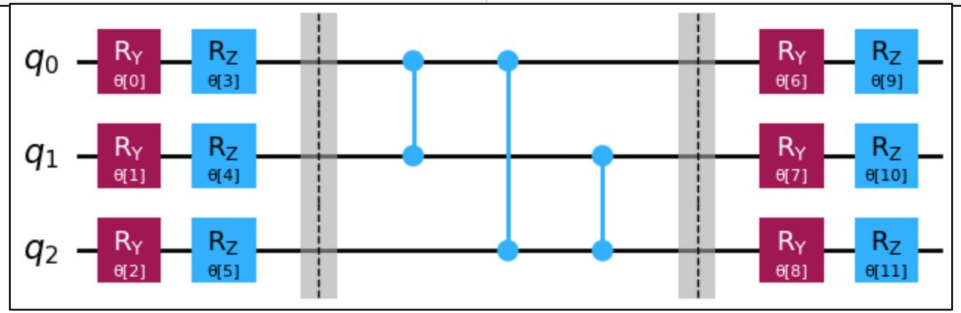
1. Setting up the Hamiltonian and ansatz (problem specification)
2. Implementing the Qiskit Runtime estimator
3. Adding the Classical optimizer and running our program

https://en.wikipedia.org/wiki/Variational_quantum_eigensolver

https://www.youtube.com/watch?v=TUFovZsBcW4

Step 1: Create a parameterized circuit to serve as the ansatz

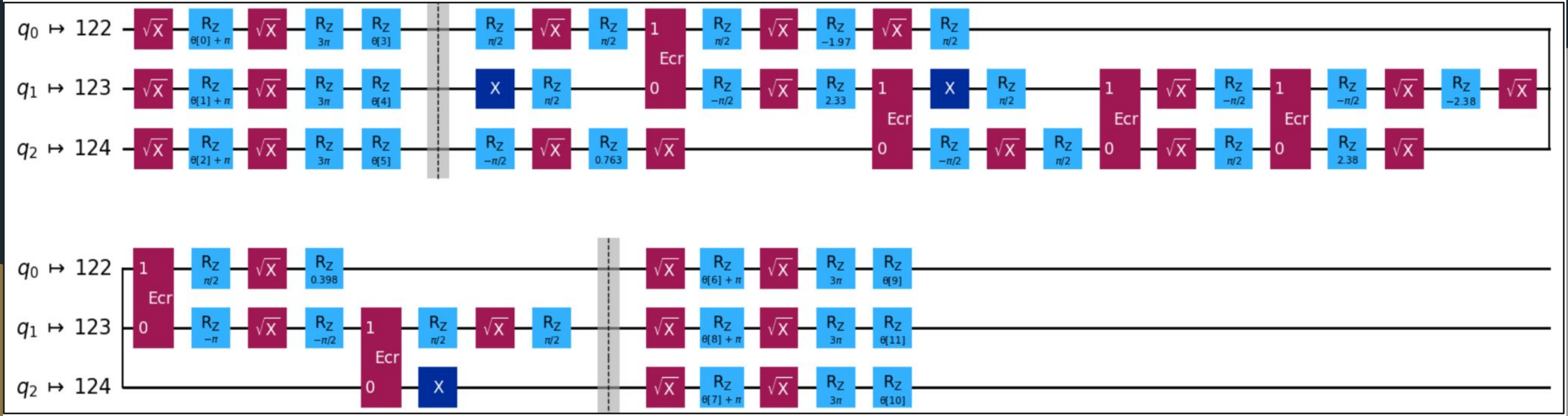
```
num_qubits = 3
rotation_blocks = ['ry','rz']
entanglement_blocks = 'cz'
entanglement = 'full'
reps = 1
insert_barriers=True
ansatz = TwoLocal(3,rotation_blocks=rotation_blocks, entanglement_blocks=entanglement_blocks,entanglement=entanglement,reps=reps,insert_barriers=insert_barriers)


### Don't change any code past this line ###
ansatz.decompose().draw('mpl')
```



Step 2: Transpile to ISA circuits

Instruction Set Architecture (ISA): The set of instructions the device can understand and execute.

```
backend_answer = FakeSherbrooke()
optimization_level_answer = 3
pm = generate_preset_pass_manager(backend=backend_answer,optimization_level=optimization_level_answer)
isa_circuit = pm.run(ansatz)
```



$q_0 \mapsto 122$: √X, Rz θ[0] + π, √X, Rz 3π, Rz θ[3], Rz π/2, √X, Rz π/2, Ecr, Rz π/2, √X, Rz −1.97, √X, Rz π/2
$q_1 \mapsto 123$: √X, Rz θ[1] + π, √X, Rz 3π, Rz θ[4], X, Rz π/2, Ecr, Rz −π/2, √X, Rz 2.33, Ecr, X, Rz π/2, Ecr, √X, Rz −π/2, Ecr, Rz −π/2, √X, Rz −2.38, √X
$q_2 \mapsto 124$: √X, Rz θ[2] + π, √X, Rz 3π, Rz θ[5], Rz −π/2, √X, Rz 0.763, √X, Ecr, Rz −π/2, √X, Rz π/2, Ecr, √X, Rz π/2, Ecr, Rz 2.38, √X
$q_0 \mapsto 122$: Ecr, Rz π/2, √X, Rz 0.398, √X, Rz θ[6] + π, √X, Rz 3π, Rz θ[9]
$q_1 \mapsto 123$: Ecr, Rz −π, √X, Rz −π/2, Ecr, Rz π/2, √X, Rz π/2, √X, Rz θ[8] + π, √X, Rz 3π, Rz θ[11]
$q_2 \mapsto 124$: Ecr, X, √X, Rz θ[7] + π, √X, Rz 3π, Rz θ[10]

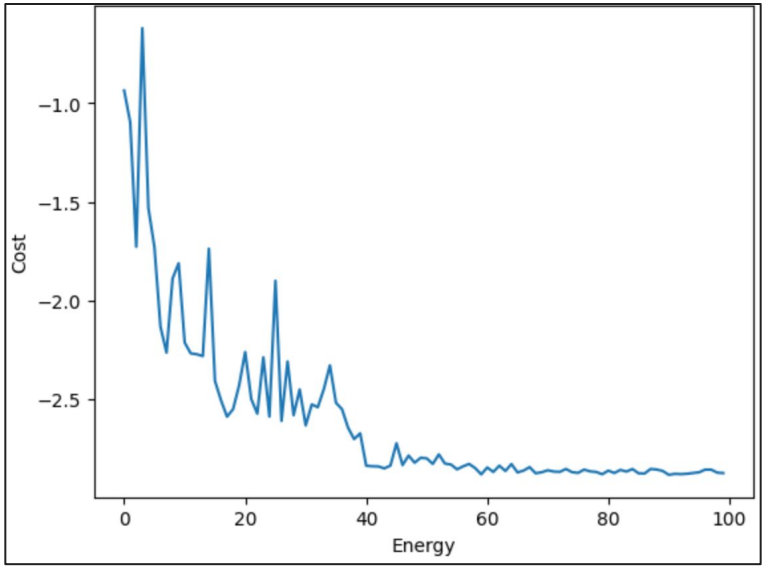
Step 3: Define a cost function

```
def cost_func(params, ansatz, hamiltonian, estimator, callback_dict):
    """Return estimate of energy from estimator

    Parameters:
        params (ndarray): Array of ansatz parameters
        ansatz (QuantumCircuit): Parameterized ansatz circuit
        hamiltonian (SparsePauliOp): Operator representation of Hamiltonian
        estimator (EstimatorV2): Estimator primitive instance

    Returns:
        float: Energy estimate
    """
    pub = (ansatz)
    result = estimator.run([(pub, hamiltonian, [params])]).result()
    energy = result[0].data.evs[0]

    callback_dict["iters"] += 1
    callback_dict["prev_vector"] = params
    callback_dict["cost_history"].append(energy)

### Don't change any code past this line ###
    print(energy)
    return energy, result
```

Step 4: Run it

```
] ### Select a Backend
  ## Use FakeSherbrooke to simulate with noise that matches closer to the real experiment. This will run slower.
  ## Use AerSimulator to simulate without noise to quickly iterate. This will run faster.

  # backend = FakeSherbrooke()
  backend = AerSimulator()

  # ### Don't change any code past this line ###

  # Here we have updated the cost function to return only the energy to be compatible with recent scipy versions (>=1.10)
  def cost_func_2(*args, **kwargs):
      energy, result = cost_func(*args, **kwargs)
      return energy

  with Session(backend=backend) as session:
      estimator = Estimator(session=session)

      res = minimize(
          cost_func_2,
          x0,
          args=(isa_circuit, hamiltonian_isa, estimator, callback_dict),
          method="cobyla",
          options={'maxiter': 100})
```

Data



Thank you!